Dino: Summary and Examples

Matthew Rosing
Robert B. Schnabel
Robert P. Weaver

CU-CS-386-88 March 1988

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

This research was supported by AFOSR grant AFOSR-85-0251, and NSF cooperative agreement DCR-8420944.

Dino : Summary and Examples

Matthew Rosing, Robert B. Schnabel, and Robert Weaver
University of Colorado at Boulder

Abstract. Dino is a new language, consisting of high level modifications to C, for writing numerical programs on distributed memory multiprocessors. Our intent is to raise interprocess communication and process control to a higher and more natural level than using messages. We achieve this by allowing the user to define a virtual machine onto which data structures can be distributed. Interprocess communication is implicitly invoked by reading and writing the distributed data. Parallelism is achieved by making concurrent procedure calls. This paper provides a summary of the syntax and semantics of Dino, and illustrates its features through several sample programs. We also briefly discuss a prototype of the language we have developed using C++.

1. Introduction. Dino ("DIstributed Numerically Oriented language") is a language for programming numerical applications on distributed memory multiprocessors, such as hypercubes. Our goal is to make parallel numerical programs for such machines easier and more natural to write and understand. We achieve this by taking advantage of the highly structured nature of the data structures, distribution of data among processors, and process structures of many parallel numerical algorithms. This enables us to raise interprocess communication and process control to a higher, more natural level than is found in most current systems.

We have previously given a brief description and motivation for the key constructs in Dino (Rosing and Schnabel [1988]). The first of these is the specification by the programmer of a virtual parallel machine that best fits the parallel algorithm. This virtual machine is referred to as a "structure of environments". Each virtual processor, or environment, is similar to an environment in a serial language. Structures of environments can be created in the form of arrays or trees. The second key construct is the ability to distribute data structures, such as arrays, over the virtual machine depending on how the data is used in the parallel algorithm. The third construct provides communications at a higher level than messages. This is achieved by reading and writing elements of a distributed data structure, utilizing the mapping of the data structure to the structure of environments to specify communication patterns. Finally, the fourth construct provides concurrent execution. This is achieved through "composite procedure calls", a form of parallel remote procedure calls. By invoking a composite procedure that is declared within an environment structure, the procedure is invoked concurrently in each environment in that structure. The combination of these

Department of Computer Science, University of Colorado, Boulder CO 80309. Research supported by AFOSR grant AFOSR-85-0251, and NSF cooperative agreement DCR-8420944.

To appear in proceedings of The Third Conference on Hypercube Computers and Applications, held in Pasadena, CA, January 1988.

constructs provide a "single program multiple data" type of parallelism.

Dino consists of modifications to the language C. These modifications could be made to most structured languages, but we have chosen C for several reasons. We have many tools for implementing a C compiler, C is available on all of our target machines, and C is a structured language which fits well with our modifications.

We have found relatively little work in this area. Languages such as Linda (Gelernter et al [1985]) and Pisces (Pratt [1987]) support numerical applications, but at a lower level than Dino. The Force (Jordan [1986]) primitives give very good support for structuring numerical applications on shared memory machines and we would like to provide similar support for distributed memory machines. Virtual machines can be declared in Structured Process (Li et al [1985]). It is possible to distribute data structures in Scott et al [1987].

Sections 2-5 of this paper paper describe the more important semantics and syntax of Dino. Section 6 briefly discusses a prototype of the language we have built using C++ (Stroustrup [1986]). In Section 7 we give four examples of Dino programs.

2. Environments. The foundation of a Dino parallel program is a data structure of virtual processors, called environments, that is constructed by the programmer to suit the parallel algorithm. An environment in Dino is similar to a block in a block structured language; it contains local data, and one or more procedures. Only one procedure can be active in an environment at one time. A single environment declaration can be used to create a data structure (usually an array) of environments. These environments are identical except for the distribution of distributed data structures among them (Section 3) and possibly a set of identifying constants. Furthermore, they can contain composite procedures (Section 5), which may be invoked from another environment, generally the host; this invocation causes the procedure to run in each environment simultaneously. This leads most naturally to a "single program, multiple data" type of parallelism. It is intended that environments can also be dynamically created to form graphs of virtual machines, but this feature is not yet implemented.

The number of environments in a Dino program can equal the number of processors on the target parallel machine, or it can be a larger number that is convenient for expressing the parallel algorithm (see Section 7). In the latter case, it is intended that the Dino compiler will combine groups of environments into a smaller number of actual processes that is appropriate for the target machine. The mapping functions for the distributed data structures provide the information that allow this to be done in a way that keeps the total amount of interprocess communication small. This feature of combining virtual processes is one of the interesting research topics associated with Dino, and it is not implemented in the current prototype.

2.1. Syntax. We will describe the syntax and semantics of Dino using examples. A non parallel "hello world" program in Dino is

```
environment host {
        main(){
                printf("hello world\n");
                } /*main*/
        } /*host*/
```

The keyword *environment* declares a scalar environment called *host* consisting of the procedure *main*. All Dino programs must have a *host* environment containing the procedure *main* as this is where execution starts. A parallel "hello world" program is:

```
environment node[2:idx][2:idy] {
        composite hello() {
                printf("hello, I am node[%d][%d]\n",idx,idy);
                } /*hello*/
        } /*node*/
```

```
environment host {
        main(){
                hello()#;
                } /*main*/
        } /*host*/
```

This program contains the scalar environment *host* and the 2×2 array of environments *node*. Each *node* has the constants *idx* and *idy* which specify where in the 2×2 array the environment is. The output of the above program might be, depending on the nondeterministic order of the print statements:

```
hello, I am node[0][0]
hello, I am node[0][1]
hello, I am node[1][0]
hello, I am node[1][1]
```

3. Distributed Data Structures. A data structure can be distributed over an environment structure in order to concurrently operate on that data structure; each environment operates on its part of the data structure. Each element of the data structure can be mapped onto one or more environments depending on how it will be used. If, for example, two environments need to access the same element of a data structure, then that element is mapped onto both environments. Such elements then can be accessed locally, where only the local copy is used, or remotely, where all of the copies are affected. The next section describes this in more detail.

The distribution of a data structure of the environment structure is defined when declaring the data structure. This distribution consists of a mapping function which can be defined by the user or selected from a library. The most common distributed data structures are single or multiple dimension arrays. The two most common classes of mapping functions that we have seen for arrays are blocking and wrapping. Blocking places consecutive rows, columns, or sub-arrays of an array onto a single environment, one block per environment. With this type of mapping it is sometimes useful to overlap the edges of the blocks; ie, the edges of the blocks are mapped to neighboring environments. Wrapping places consecutive rows, columns, or sub-arrays onto consecutive environments, and then repeats this pattern until the entire data structure has been mapped. This is most useful for load balancing. The details of defining mapping functions will not be discussed here.

Dynamic distributed data structures are planned but will not be discussed here.

3.1. Syntax. The declaration of distributed matrices used for matrix multiplication could be

```
environment node[N:id] {
        distributed float A[N][N] : byRow;
        distributed float B[N][N] : byCol;
        ...}
```

which would declare N environments *node*, and place one row of the matrix A and one column of B on each environment. The identifiers *byRow* and *byCol* are mapping functions which were previously defined by the user or are in a library. An example of an application using a one to many mapping is the solution of Poisson's equation:

```
environment node[N:idx][N:idy]{
        distributed float U[N+2][N+2] : NSEWoverlap;
        ... }
```

An $N \times N$ array of environments *node* is declared, and Each element of U is mapped to the corresponding element of *node* and its four nearest neighbors to the north, south, east, and west. (The matrix U is larger than *node* because of the border points.) Both of these examples are completed in Section 7.

4. Communication. Communication between environments is accomplished by reading and writing distributed variables. Distributed variables can be accessed locally or remotely. A local access to a distributed variable is the same as a standard access to a standard, local variable (the distributed element accessed must be mapped to the environment accessing it). A remote access is used to generate communications. A remote write to a variable causes a message, containing the value, to be sent to all of the other environments to which the variable is mapped. A corresponding read of the variable on one of the other environments receives the message and assigns the value to the variable. This is how multiple copies of the same variable are kept consistent across many environments. Remotely accessing a variable which is mapped only to the current environment is the same as doing a local access. It is possible to write to or read from a distributed variable that is not mapped to one's environment; this is primarily useful for shifting data, such as the rotate procedure in the matrix multiplication example.

Synchronization in Dino is accomplished by reading and writing distributed variables (and by the barrier implicit at the end of a composite procedure call). Distributed variables may be accessed either synchronously or asynchronously; the synchronization attribute is associated with the data structure, which can be declared as either "distributed" (which is the default and implies synchronous access) or "asynch distributed." A remote read to a synchronous variable will used the first unread value of this variable that has been received from another environment; if there is none it will block until another environment does a write. Synchronous remote writes do not block. A remote read to an asynchronous variable will use the last value received, if any, otherwise a local read is made. A remote write to an asynchronous variable is the same as a remote synchronous write. Note that the style of data synchronization in a Dino program can be altered by simply changing the declaration of distributed variables.

4.1. Syntax. The syntax of a local access to a distributed variable is the same as in C. The syntax of a remote access is the same as a local access except that the character '#' follows the reference. An example of this is shown in a section of the program solving Poisson's equation. Assuming the definition given in the previous example, the following statement would do the actual calculation on *node* $[x-1][y-1]$ (offsets used because of border):

```
U[x][y]# = U[x][y] + (U[x+1][y]# + U[x-1][y]# + U[x][y+1]# + U[x][y-1]# - 4*U[x][y]) * w/4;
```

The right hand expression consists of a local read of $U[x][y]$ and a remote read of $U[x+1][y]$, $U[x-1][y]$, $U[x][y+1]$, and $U[x][y-1]$. The computation waits, if necessary, until those values are received from the four neighboring environments, calculates the new value of $U[x][y]$, and then broadcasts the new value to the neighbors.

5. Parallel Execution. Concurrency in Dino is achieved by using a composite procedure call, a form of parallel remote procedure calls. A "composite procedure" consists of a set of identical procedures with one procedure in each environment of an environment structure. Invocation of a composite procedure calls all of the procedures simultaneously. Each procedure can work on data which is resident in its environment, or on data that has been passed to it as a parameter. Parameters can be standard variables, which are replicated and sent to each procedure, or they can be distributed variables, which are broken up and distributed based on their mapping function. Unlike C, entire arrays can be passed as parameters. Furthermore, parameters can be passed by value, result, or both.

5.1. Syntax. A composite procedure must be declared within an environment, and is preceded by the keyword *composite*. The formal parameters are preceded by the keywords *in* or *out* if they are passed by value or result respectively. The default is pass by value/result. An example for Poisson's equation, in which the initial array U is passed as a parameter to the composite procedure *Poisson*, would be:

```
environment node[N:idx][N:idy]{
        composite Poisson(U, in itns)
                distributed float U[N+2][N+2] : NSEWoverlap;
                int itns;  /*number of iterations*/
                { ... }
        }
```

The array *U* is passed by value/result and is distributed to each procedure of *Poisson* based on the mapping function *NSEWoverlap*. Integer *itns* is replicated and passed to each procedure, by value only. Invocation of this procedure could be done from the following environment:

```
environment host {
        float V[N+2][N+2];
        main() {
                initV(V);
                Poisson(V[][],250)#;
                }
        }
```

The '#' indicates that a remote procedure call is made. Furthermore, the syntax *V*[][] selects the entire array *V*. The ability to move subsections of arrays in a single statement is used quite often and therefore Dino provides convenient facilities for assignments and parameter passing to be done on subsections of arrays.

6. Dino/C++. We have developed a prototype of Dino using C++ (Stroustrup [1986]). C++ is an object-oriented extension of C. Its object oriented nature is useful for simulating language constructs. C++ classes are used to implement environments, composite procedures, distributed data structures, and mapping functions. We have built only a subset of Dino as certain language features are difficult to implement without a compiler. The subset includes one and two dimensional arrays of environments, and distributed one or two dimensional arrays of integer or double precision numbers. The mapping functions include block, block with overlap, and one to all. Finally, there are several differences in the syntax between Dino and Dino/C++.

The prototype runs on our network of Sun work stations and on an Intel hypercube. Our experience is that it is much easier to write programs using Dino/C++ than using the libraries provided with these machines. We would not, however, use Dino/C++ for writing a large application because the resulting code would be very slow. One of the major causes of this is that much of the work that should be done at compile time is done at run time in Dino/C++. Another inefficiency occurs in the message passing system. There is a lot of overhead in copying and packaging messages before handing them off to the underlying system. We believe that these problems can be corrected when a full compiler for Dino is written.

7. Examples. This section contains four examples of Dino programs. We believe that they illustrate that Dino permits simple representations of parallel numerical algorithms that correspond closely to the naturally way one would informally describe that parallel algorithm.

Note that in the first example, the number of environments is representative of the number of processors on a parallel machine, while in the other three examples the number of environments is chosen to naturally reflect the data parallelism in the problem, and may be larger than the number of actual processors.

7.1. Example 1: Dot Product. This is a simple program illustrating parameter passing to composite procedures. In this example the dot product of two vectors are calculated. The vectors are distributed in block fashion over the node environments. Each node environment calculates the "partial" dot product and returns the result to the host which sums up the partial dot products. The partial product of *node*[*i*] is stored in *z* [*i*].

```
#define N 1024
#define P 32

environment node[P:id] {
        composite dotProd(in x, in y, out z)
                distributed float x[N] : block;
                distributed float y[N] : block;
                distributed float z[P] : element;
                {
                int i;
                z[id] = 0;
                for (i=id*N/P; i<(id+1)*N/P; i++)
                        z[id] += x[i]*y[i];
                }
        }

environment host{
        main(){
                float x[N], y[N], z[P];
                int i;
                float sum;

                init(x,y);
                dotProd(x[], y[], z[])#;
                for (sum=i=0; i<P; i++)
                        sum += z[i];
                printf("sum is %f\n",sum);
                }
```

7.2. Example 2: Poisson's equation. This example illustrates the use of a distributed array using a one to many mapping. Each element $U[x][y]$ is mapped to *node* [*xid*][*yid*] and the four neighboring environments to the north, south, east, and west. The communication between these environments is implicit in the reads and writes to U, due to the mapping function. Each composite procedure reads the values from the neighboring environments (waiting if they have not already arrived), calculates the new value of $U[x][y]$, and broadcasts the value to the four neighbors. The algorithm is a red-black SOR method (see eg Adams and Jordan [1986]).

```
#define N 128

environment node[N:xid][N:yid]{

        composite Poisson(U, in itns)
                distributed float U[N+2][N+2]:NSEWoverlap;
                int itns; /*number of iterations*/
                {
                int i,x,y,w;

                x = xid + 1;
                y = yid + 1;
```

```
            w = 1.8; /*relaxation factor*/
                    /*if red block: calculate using local values to start cycle*/
            if ( (x+y)%2)
                    U[x][y]# = U[x][y] + (U[x+1][y] + U[x-1][y] + U[x][y+1] +
                            U[x][y-1] -4*U[x][y]) * w/4;
                    /*calculate remaining iterations using remote values*/
            for (i=1; i<itns; i++)
                    U[x][y]#  = U[x][y] + (U[x+1][y]# + U[x-1][y]# +
                            U[x][y+1]# + U[x][y-1]# -4*U[x][y]) * w/4 ;
            } /*poisson*/
    } /*node*/

environment host{
    float G[N+2][N+2];

    main(){
            init(G);
            Poisson(G[][], 250)#;
            display(G);
            } /*main*/
    }/*host*/
```

7.3. Example 3: Matrix Multiplication. This program, which multiplies two matrices together, illustrates a more complicated environment, and how distributed data structures can be passed as parameters to non-composite procedures. The composite procedure *multiply* takes the matrices *a* and *b* and distributes them by row and by column onto the *node* environments. Each node starts by calculating *c*[*id*][*id*]. The *b* matrix is then rotated to the left after which each node calculates c[id][id+1]. This is repeated for N cycles.

Note that *rotate*, which is a "normal" procedure has the formal parameter *M* which is a distributed variable. The mapping function is that of the actual parameter. Only a reference to the array is passed, not the entire array.

```
#define N 1024
#define LEFT(id) id>1?id-1:N-1
#define RIGHT(id) id<N-1?id-1:0

environment node[N:id]{
    rotate(M) /*rotate the b matrix to the left*/
            distributed float M[N][N];
            {
            M[][LEFT(id)]# = M[][id];
            M[][id] = M[][RIGHT(id)]#;
            }

    composite multiply(in a, in b, out c)
            distributed float a[N][N] :byRow;
            distributed float b[N][N] :byCol;
            distributed float c[N][N] :byRow;
            {
            int cycle,k;
            float sum;
```

```
                for (cycle=0; cycle<N; cycle++){ /*for each element of c[id][]*/
                        for (k=sum=0; k<N; k++) /*generate dot product*/
                                sum += a[id][k] * b[k][id];
                        c[id][(cycle+id)%N] = sum;
                        rotate(b); /*rotate the b matrix*/
                        } /*for*/
                } /*multiply*/
        } /*node*/

environment host{
        float at[N][N]; /*temporary matrices*/
        float bt[N][N];
        float ct[N][N];

        main(){
                init_data(at,bt);
                multiply(at[][],bt[][],ct[][])#;
                print_matrix(ct);
                }
        }
```

7.4. Example 4: Gaussian Elimination. This program illustrates the use of an environment with several composite procedures. The equation Ax=b is solved using Gaussian elimination with partial pivoting and two back solves. The node environments consist of the composite procedures *gausElim* , *lyb* , *uxy* , the distributed data structures A and y, and the replicated data structure p. The pivot vector p is interesting because each environment maintains a complete, up-to-date copy. (Other implementations are possible.) For brevity, the functions *uxy* and *findPivot* are stubs.

The procedure *gausElim* takes as input the A matrix and calculates the PLU decomposition. A is distributed by columns onto the node environments. This makes pivoting simpler. The algorithm works as follows. For each of N cycles, the process holding the current pivot column determines the pivot row and broadcasts it, and each process pivots its column. Then the pivot process determines the multipliers and broadcasts them, and each processor to the right of the pivot column eliminates its column. Note that the vector of multipliers is sent in a single statement; the syntax $m[<diag, N>]$ signifies the subvector of m from *diag* up to and not including N.

The procedure *lyb* calculates the solution y to $Ly=Pb$, while *uxy* calculates the desired solution x by solving $Ux=y$. The procedure *lyb* executes as follows: for each $y[i]$, in order, it calculates the vector ly which is A[j][i]*y[i], j>i. It distributes this vector to the other processors, which then subtract $ly[id]$ from $y[id]$.

```
#define N 128

environment col[N:id]{ /*one env for each column*/
        distributed float A[N][N]:byCol;
        distributed float y[N]  :element;
        int p[N];               /*saves pivot order*/
        int findPivot(){
                /*finds pivot row on this column*/
                }

        composite gausElim(in At)
                distributed float At[N][N]:byCol;
```

```
            {
            distributed float m[N] : ALL;  /*multipliers for eliminating columns*/
            distributed int pRow   : ALL;  /*the pivot row*/
            int diag, i;                  /*diag is the current column eliminated*/

            A[][id] = At[][id];            /*transfer At to a global position*/
            for (i=0; i<N; i++)            /*init pivot order*/
                    p[i] = i;
            for (diag=0; diag<N-1; diag++){ /*for each  column to eliminate*/
                    if (diag==id){
                            pRow# = findPivot();        /*broadcast the pivot row*/
                            swap( &p[pRow], &p[diag]);  /*pivot 'b' vector*/
                            swap( &A[pRow][id], &A[diag][id]); /*pivot A*/
                            for (i=diag+1; i<N; i++)    /*determine multipliers*/
                                    A[i][id] /= A[id][id];
                            m[<diag,N>]# = A[<diag,N>][id]; /*send multipliers*/
                            }
                    else {
                            swap( &p[pRow#], &p[diag]);  /*recv pRow and pivot 'b' vector*/
                            swap( &A[pRow][id], &A[diag][id]); /*pivot A*/
                            m[<diag,N>] = m[<diag,N>]#;  /*receive multipliers*/
                            if (diag<id)              /*eliminate column*/
                                    for (i=diag+1; i<N; i++)
                                            A[i][id] -= m[i]*A[diag][id];

                            }
                    }
            }

    composite lyb(in b) /*solve ly=b for y*/
            distributed float b[N]  : ALL;
            distributed float ly[N] : block;  /*l[i] =  y[j]*A[i][j]*/
            int i,j;

            y[id] = b[p[id]];
            for (i=0; i<N; i++) {          /*for each row*/
                    if (i==id){              /*calculate y[i]*/
                            for (j=id+1; j<N; j++)
                                    ly[j]# = y[id]*A[j][id]; /*send product where it is needed*/
                            }
                    else if (i<id)       /* y[id] == */
                            y[id] -= ly[id]#; /*b[p[id]] - sum of y[i]*A[i][id]; 0<i<diag*/
                    }
            }

    composite uxy(out x)
            distributed float x[N] : ALL;
            {} /* ... solve Ux=y ...*/

environment host{
    init(A,b)
            {} /*... initialize A and b arrays ...*/
```

```
main(){
        float A[N][N];
        float b[N];
        float xt[N],x[N];
        int p[N],i;

        init(A,b);
        gausElim(A[][])#;               /*calculate LU*/
        lyb(b[])#;                  /*solve Ly=b for b*/
        uxy(xt[],p[])#;              /*solve Ux=y for x*/
        for (i=0; i<N; i++) x[i] = xt[p[i]];  /*un pivot x vector*/
        }
}
```

REFERENCES

(1) D. GELERNTER, N.CARRIERO, S. CHANDRAN, and S.CHANG, *Parallel programming in Linda,* in Proceedings of the 1985 International Conference on Parallel Processing, IEEE Press, 1985, pp. 255-263.

(2) H. F. JORDAN, *Structuring parallel algorithms in an MIMD, shared memory environment,* Parallel Computing 3, 1986, pp. 93-110.

(3) T. PRATT, *The Pisces 2 parallel programming environment,* in Proceedings of the 1987 International Conference on Parallel Processing, IEEE Press, 1987, pp. 439-445.

(4) B. STROUSTRUP, *The C++ Programming Language,* Addison-Wesley, Reading, Massachusetts, 1986.

(5) L. SCOTT, J. BOYLE, and B. BAGHER, *Distributed Data Structures for Scientific Computation,* in 1986 Proceedings of the Second Conference on Hypercube Multiprocessors, SIAM, 1987, pp. 55-66.

(6) H. LI, C. WANG, and M LAVIN, *Structured Process,* in Proceedings of the 1985 International Conference on Parallel Processing, IEEE Press, 1985, pp. 247-254.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CS-CU-385-88	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION University of Colorado	6b. OFFICE SYMBOL *(If applicable)*
7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research/NM	
6c. ADDRESS *(City, State and ZIP Code)* Computer Science Department Campus Box 430 Boulder, CO 80309-0430	7b. ADDRESS *(City, State and ZIP Code)* Building 410 Bolling Air Force Base, DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL *(If applicable)*
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER AFOSR-85-0251	
8c. ADDRESS *(City, State and ZIP Code)*	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.

11. TITLE *(Include Security Classification)*
Dino: Summary and Examples

12. PERSONAL AUTHOR(S)
Matthew Rosing, Robert B. Schnabel, Robert P. Weaver

13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM ________ TO ________	14. DATE OF REPORT *(Yr., Mo., Day)* 88/3/1	15. PAGE COUNT 10

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB. GR.	Parallel programming language, numerical computation, distributed memory multiprocessor

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Dino is a new language, consisting of high level modifications to C, for writing numerical programs on distributed memory multiprocessors. Our intent is to raise interprocess communication and process control to a higher and more natural level than using messages. We achieve this by allowing the user to define a virtual machine onto which data structures can be distributed. Interprocess communication is implicitly invoked by reading and writing the distributed data. Parallelism is achieved by making concurrent procedure calls. This paper provides a summary of the syntax and semantics of Dino, and illustrates its features through several sample programs. We also briefly discuss a prototype of the language we have developed using C++.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL Brian W. Woodruff, Major USAF	22b. TELEPHONE NUMBER *(Include Area Code)* 202/767-5025	22c. OFFICE SYMBOL

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE